Tame .NET Events

Take advantage of .NET's event-handling options to improve on the basics or craft powerful event-handling solutions.

by Juval Löwy

Technology Toolbox

- ☐ VB.NET
- ✓ C#
- ☐ SQL Server 2000
- ☐ ASP.NET
- ☐ XML
- ☐ VB6

Almost every application requires some form of events publishing and subscription mechanism. An event is usually nothing more than an object (the "source" or "publisher") notifying all interested parties (the "sinks" or "subscribers") about something that took place on the object's side. You accomplish this notification by calling methods on the sink objects. .NET makes this formerly tough development task a lot easier. This article discusses .NET events design guidelines and development tricks that promote loose coupling between publishers and subscribers, improve availability, conform to existing conventions, and generally take advantage of .NET's rich events support infrastructure (see Resources for more information about delegates and events).

Before .NET, you were on your own, forced to handcraft event-handling solutions, often reinventing the wheel and repeating mundane event-handling code over and over. The source of the problem was that development languages such as C++ or component technologies such as COM were event-oblivious: Nothing in the language or the component technology supported events inherently. You had to add that support on top, using function pointers in C and C++ or COM's connection points protocol. The resulting programming model was usually cumbersome and messy, and it often coupled the event publisher to the event subscriber. .NET, on the other hand,

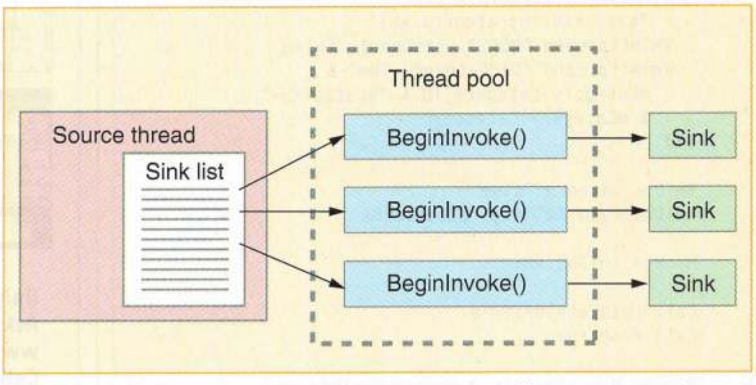

Figure 1 Call Asynchronous Events With BeginInvoke(). By using BeginInvoke() to fire an event, the event to each sink gets fired on a different thread from the thread pool. That way, if one or more of the event's target sinks take a while to process the the event, or even hang, the rest of the source process isn't brought to its knees waiting.

Resources

C# Explorer, "Delve Into Delegates" by Steve Lardieri [*VSM* September 2001] treats events as first-class citizens, with standard out-of-the-box support for connection setup and tear-down, management of the subscribers list, and of course, the publishing act itself.

.NET events rely on delegates: type-safe method references and overloaded operators used to manage a connection. A delegate defines only the signature of the method the sinks should provide to receive a specific event on. Although technically the delegate can define any method signature, in practice you should conform to a specific signature. The arguments go like this: First, the method should have a void return type only. It doesn't make any sense to return a value because the publisher has no need for values. The publisher has no idea why a subscriber wants to subscribe in the first place.

In addition, the delegate class hides the actual publishing act from the publisher. The delegate is the one iterating over the sink list, calling each corresponding method. The returned values aren't propagated to the publisher's code. Also, some subscribers would likely want to receive the same event from multiple sources. Because there's no flexibility in providing as many methods as publishers, the subscriber wants to provide the same method to multiple publishers. However, the signature should contain the publisher's identity so the subscriber can distinguish between different publishers. The easiest and most generic way to do this is to add a parameter of type *object*, called the *sender parameter*:

```
public delegate void
  NumberChangedDelegate(
object sender,int num);
```

Publishers can simply pass themselves as the sender (using the reserved words *this* in C# or *Me* in VB.NET).

Couple Publishers to Subscribers

Finally, defining actual event arguments such as *int num* couples publishers to subscribers, because the subscriber has to expect a particular set of arguments. Changing these arguments in the future affects all subscribers. To contain an argument change, it's better to provide a generic event arguments object as a parameter. .NET provides the EventArgs class to do this:

```
public class EventArgs
{
   public static readonly
      EventArgs Empty;
   static EventArgs()
   {
      Empty = new EventArgs();
   }
   public EventArgs(){}
}
```

Pass in an EventArgs object instead of specific event arguments:

```
public delegate void
  NumberChangedDelegate(
    object sender,
    EventArgs eventArgs);
```

C# • Prevent Subscribers From Modifying Event Arguments

```
public class NumberEventArgs1 : EventArgs
{
   public readonly int Num;
   NumberEventArgs1(int num)
   {
   Num = num;
   }
}
public class NumberEventArgs2 : EventArgs
{
   protected int m_Num;
   NumberEventArgs2(int num)
   {
      m_Num = num;
   }
   int Num
   {
      get
      {
         return m_Num;
      }
   }
}
```

Listing 1 By default, one subscriber can affect all other subscribers that handle the event after it by changing the event arguments. You can prevent subscribers from modifying the event arguments either by using a read-only property or applying the *readonly* access modifier to a public member.

If the publisher has no need for an argument, pass in EventArgs.Empty, taking advantage of the static constructor and the read-only Empty class member.

If the event requires arguments, derive a class from EventArgs, such as MyEventArgs; add member variables, methods, or properties as required; and pass in MyEventArgs. The subscriber should downcast the generic EventArgs to the specific argument class associated with this event (MyEventArgs, in this example) and access the arguments. Doing so allows you to add arguments, remove unused arguments, derive yet another class from MyEventArgs, and so on, without forcing a change on the subscribers.

The resulting delegate definition is so generic that .NET already defines such a delegate—the EventHandler delegate:

```
public delegate void
  EventHandler(
    object sender,
    EventArgs eventArgs);
```

As a result, you don't need to define your own delegates for events. You should use EventHandler in almost all cases, although the rest of this article doesn't use EventHandler for simplicity's sake. A bit more information about signatures: The convention for the sink method name is On<EventName>, making the code standard and readable.

As explained previously, you should provide event arguments in a class derived from EventArgs, and have the arguments as class members. The delegate class iterates over its list of subscribers, passing the argument object from one subscriber to the next.

However, nothing prevents a particular subscriber from modifying the argument values and affecting all other subscribers that handle the event after it. Usually, you should prevent subscribers from modifying these members as the event delegate passes the event argument object from one subscriber to the next. To preclude changing the members, either provide access to them as read-only properties or expose them as public members and apply the *readonly* access modifier. In both cases, you should initialize the argument value in the constructor (see Listing 1).

The publisher should always check the delegate for a null value before attempting to publish. The reason is that if nobody has subscribed to the event, the delegate list is empty, and .NET throws an exception when trying to access it. Speaking of exceptions, the publisher has no way of knowing how disciplined the subscribers are. Some subscribers might encounter an exception in their event handling, not catch it, and cause the publisher to crash. Always publish inside a try/catch block:

```
public class MySource
{
   public event EventHandler m_MyEvent;
   public void FireEvent()
   {
      try
      {
         if(m_MyEvent != null)
         m_MyEvent(this,
         EventArgs.Empty);
      }
      catch
      {
         //handle exceptions
      }
   }
}
```

Event accessors provide a similar benefit to properties: They hide the actual class member while maintaining the original ease of use.

Earlier, you saw that you access the delegate member variable directly to hook up a subscriber to a publisher. Exposing class members in public is asking for trouble. Doing so violates the core object-oriented principle of encapsulation and information hiding, and couples all the subscribers to the exact member variable definition. .NET overcomes this issue by providing a property-like mechanism, called event accessors. Accessors provide a similar benefit to properties: They hide the actual class member while maintaining the original ease of use. Event accessors use language-specific reserved words. C# uses *add* and *remove* to encapsulate the

C# • Encapsulate Event Delegate Member Variables

```
public delegate void NumberChangedDelegate(int
   num);

public class MySource
{
   protected event NumberChangedDelegate
      m_NumberChangedEvent;
      public event NumberChangedDelegate
         NumberChangedEvent
      {
         add
         {
            m_NumberChangedEvent += value;
         }
         remove
         {
            m_NumberChangedEvent -= value;
         }
      }
      public void FireEvent(int num)
      {
         m_ NumberChangedEvent(num);
      }
   }

MySource source = new MySource();
MySink sink = new MySink();

//Setup connection:
source.NumberChangedEvent += new
   NumberChangedDelegate(sink.OnNumberChanged);
//Fire Event
source.FireEvent(42);
//Teardown connection:
source.NumberChangedEvent -= new
   NumberChangedDelegate(sink.OnNumberChanged);
```

Listing 2 Event accessors encapsulate the actual event delegate member variable (note the "protected" modifier). Accessors allow the same client code as they do with raw public event delegate members.

C# • Manage Large Numbers of Events

```
using System.ComponentModel;

EventHandlerList eventList;
eventList = new EventHandlerList();

//To add to the list:
eventList.AddHandler(m_Button1,new
   EventHandler(OnButtonClicked));

//To fire event:
EventHandler handler =
(EventHandler)eventList[m_Button1];
handler(m_Button1,EventArgs.Empty);

//To remove from the list:
eventList.RemoveHandler(m_Button1,new
   EventHandler(OnButtonClicked));
```

Listing 3 Use the EventHandlerList class when it's impractical to dedicate member variables per event. EventHandlerList stores value/key pairs identifying the event and its handler. You can use any .NET object as a key.

C# • Manage Sink Interfaces

```
public class MySource
{
  protected event EventHandler m_event1;
  protected event EventHandler m_event2;
  protected event EventHandler m_event3;
  public void Advise(IMySink sink,EventType
    eventType)
  {
    if((eventType & EventType.OnEvent1) ==
      EventType.OnEvent1)
    {
      m_event1 += new
        EventHandler(sink.OnEvent1);
    }
    if((eventType & EventType.OnEvent2) ==
      EventType.OnEvent2)
    {
      m_event2 += new
        EventHandler(sink.OnEvent2);
    }
    //if EventType.OnEvent3…
  }
  public void Unadvise(IMySink
sink,EventType
    eventType)
  {
    if((eventType & EventType.OnEvent1) ==
      EventType.OnEvent1)
    {
      m_event1 -= new
        EventHandler(sink.OnEvent1);
    }
    if((eventType & EventType.OnEvent2) ==
      EventType.OnEvent2)
    {
      m_event2 -= new
        EventHandler(sink.OnEvent2);
    }
    //if EventType.OnEvent3…
  }
  public void FireEvent(EventType
eventType)
  {
    if((eventType & EventType.OnEvent1) ==
      EventType.OnEvent1)
    {
      m_event1(this,EventArgs.Empty);
    }
    if((eventType & EventType.OnEvent2) ==
      EventType.OnEvent2)
    {
      m_event2(this,EventArgs.Empty);
    }
    //if EventType.OnEvent3…
  }
}
```

Listing 4 The publisher can further encapsulate the actual event implementation by providing methods that manage connection to sink interfaces. This schema also saves round trips and promotes loose coupling between the publisher and its subscribers.

actual event member variable, performing the += and -= functions respectively (see Listing 2).

Manage Numerous Events

Having a class that deals with a large number of events is common when developing frameworks. For example, the Control class in the System.Windows.Forms namespace has events corresponding to most Windows messages—a huge number, by any account. The problem in such cases is that it's impractical to allocate a class member and/or accessor per event. The class definition is unmanageable, not to mention problems that occur with documentation, CASE tool support, and even IntelliSense. .NET provides the EventHandlerList class (defined in the System.ComponentModel namespace) just for this predicament. EventHandlerList is a linear list that stores value/key pairs. The key is a generic object identifying the event, and the value is an EventHandler instance. The key can be an index, a string, a particular button, and so on (see how to use the EventHandlerList class in Listing 3).

This class adds the OnButtonClicked() method as an event handler to the m_Button1 button class member, fires the event, and removes the event handler from the list. Note the key here is an instance of an object (a button), but it could be anything else—for example, a string such as "m_Button1".

Event accessors provide just enough encapsulation by hiding the actual event members. However, you can improve on this model. For one, a subscriber might want to subscribe to a set of events. The event subscriber shouldn't have to make multiple, potentially expensive calls, both to set up and tear down the connections. The subscriber doesn't need know about the event accessors in the first place, and the subscriber might want to receive events on an entire sink interface instead of individual methods. The next step is to provide a simple but generic way to manage connections, one that saves the redundant calls, encapsulates the event accessors and members, and allows objects to sink interfaces.

Imagine an interface that defines a set of events, the IMySink interface:

```
public interface IMySink
{
  void OnEvent1(
     object sender,
     EventArgs eventArgs);
  void OnEvent2(
     object sender,
     EventArgs eventArgs);
  void OnEvent3(
     object sender,
     EventArgs eventArgs);
}
```

Anybody can implement this interface, and the interface is all the publisher should know about.

Next, define an enumeration of the events, with an enum for each event. Mark the enum with the [Flags] attribute:

```
[Flags]
public enum EventType
{
  OnEvent1,
  OnEvent2,
  OnEvent3,
  OnAllEvents =
```

C# • Fire Events Asynchronously

```
public delegate void NumberChangedDelegate(int
   num);

public class MySource
{
    public event NumberChangedDelegate
      m_NumberChangedEvent;

    public void FireEventAsynch(int num)
    {
      Delegate[] delegates =
         m_NumberChangedEvent.
         GetInvocationList();
      foreach(Delegate del in delegates)
      {
         NumberChangedDelegate sink =
           (NumberChangedDelegate)del;
         sink.BeginInvoke(num,null,null);
      }
    }
    public void FireEvent(int num)
    {
      m_ NumberChangedEvent(num);
    }
}
```

Listing 5 You can't call BeginInvoke() on an event delegate, because the delegate will probably have more than one callback. Instead, use GetInvocationList() to access the callback list, and call BeginInvoke() on each sink delegate yourself.

```
   OnEvent1|OnEvent2|OnEvent3
}
```

The [Flags] attribute indicates you can use the enum values as a bit-mask (see the way EventType.OnAllEvents is defined).

The publisher provides two methods, Advise() and Unadvise(), that accept two parameters: the interface and a bit-mask flag indicating which events to subscribe the sink interface to. Internally, the publisher could have an event delegate member per method on the sink interface, or just one for all methods (it's an implementation detail, so the subscriber shouldn't care). Advise() checks the flag and subscribes the corresponding interface method, and Unadvise() unsubscribes (see Listing 4, which also shows the FireEvent() method, with error handling removed for clarity). The example in Listing 4 uses an event member variable per method on the sink interface.

The code required to advise or unadvise is equally straightforward, yet it shows the elegance of this approach for sinking whole interfaces with one call, and how completely encapsulated the actual event class members are:

```
MySource source = new MySource();
IMySink sink =
new MySink();
//subscribe to events 1 and 2:
source.Advise(sink,
EventType.OnEvent1 |
   EventType.OnEvent2);
```

Fire Events Asynchronously

Regardless of how you manage your events, the event is blocked when the publisher fires it until all subscribers finish handling the event. Only then does control return to the publisher. Disciplined and well-behaved subscribers shouldn't perform any lengthy operation in their event handlers. The problem is, the publisher can't tell whether it's dealing with disciplined subscribers. The solution is to fire the events asynchronously.

In the past, developers created worker threads to publish on, which you can still do in .NET. However, .NET has built-in support for asynchronous method invocation. The delegate is actually a sophisticated class, and it has an asynchronous BeginInvoke() method. BeginInvoke() accepts the same parameters as the delegate itself, as well as two other parameters used to retrieve returned values from the method and notify when it's completed. These parameters are of little use when publishing events—there shouldn't be any returned parameters, and the publisher shouldn't care when the subscriber is done processing the events. BeginInvoke() uses a thread from the thread pool to dispatch the call, and returns immediately (see Figure 1). Unfortunately, you can't call BeginInvoke() on the event member directly. You can invoke BeginInvoke() only if the delegate's internal list of sink callbacks (actually other delegates) has only one target in it. If you have more than one, the delegate throws an exception:

```
//this throws an exception:
public void FireEventAsynch(int num)
{
   m_NumberChangedEvent.
   BeginInvoke(num,null,null);
}
```

The workaround is to iterate over the event delegate internal invocation list, calling BeginInvoke() on every one of them. You access the internal list using the GetInvocationList() method (see Listing 5 for how to fire the same event as in Listing 2 asynchronously).

Taming .NET events is essential for any decent-size application to reduce coupling between event publisher and subscribers, and to promote encapsulation and ease of maintenance. This code improves on the raw technology offered by .NET; its programming techniques are applicable to almost all parts of .NET, and are valuable tools in your development toolbox. **VSM**

Juval Löwy is a seasoned software architect and the principal of IDesign, a consulting and training company focused on .NET design and migration. Juval's the author of *COM and .NET Component Services* (O'Reilly & Associates). This article is based on excerpts from his upcoming book on developing .NET components. Contact him at www.bartonsphere.net/idesign.

Go Online

Use these Locator+ codes at www.visualstudiomagazine.com to go directly to these related resources.

VS0204 Download all the code for this issue of *VSM*.

VS0204JL Download the code for this article separately. This article's code includes a Visual Studio .NET solution with independent projects demonstrating each key point in the article.

VS0204JL_T Read this article online.